MICROSAR WDGIF

Technical Reference

Version 1.2.0

Authors	Christian Leder, Rene Isau
Status	Released

Document Information

History

Author	Date	Version	Remarks
Christian Leder, Rene Isau	2016-03-16	1.0.0	First version of the migrated WdgIf Technical Reference
Christian Leder	2016-07-13	1.1.0	Update after introduction of native CFG5 generator
Christian Leder	2017-01-09	1.2.0	Update after removing state combiner automatic mode

Reference Documents

No.	Source	Title	Version
[1]	AUTOSAR	AUTOSAR_SWS_WatchdogInterface.pdf	V2.3.0
[2]	Vector Informatik	Safety Manual	
[3]	AUTOSAR	AUTOSAR_TR_BSWModuleList.pdf	V1.4.0

Caution
We have configured the programs in accordance with your specifications in the questionnaire. Whereas the programs do support other configurations than the one specified in your questionnaire, Vector´s release of the programs delivered to your company is expressly restricted to the configuration you have specified in the questionnaire.

Contents

Illustrations

Tables

1 Component History

The component history gives an overview over the important milestones that are supported in the different versions of the component.

Component Version	New Features
1.00	Migration of the WdgIf to Vector Informatik GmbH
2.00	Introduction of native CFG5 generator
2.01	Removing manual state combine mode

Table 1-1 Component history

2 Introduction

This document describes the functionality, API and configuration of the AUTOSAR BSW module WdgIf as specified in [1].

Supported AUTOSAR Release*:	4.0.1	
Supported Configuration Variants:	pre-compile	
Vendor ID:	WDGIF_VENDOR_ID	30 decimal (= Vector-Informatik, according to HIS)
Module ID:	WDGIF_MODULE_ID	43 decimal (according to ref. [3])

* For the detailed functional specification please also refer to the corresponding AUTOSAR SWS.

This user manual describes the Watchdog Interface (WdgIf), which is part of the Watchdog Manager Stack, which is part of the AUTOSAR ECU Abstraction Layer. The main WdgIf functionality consists of linking one or more Watchdog drivers (Wdg) to the overlying Watchdog Manager module (WdgM).

For multi-core systems, the WdgIf additionally offers the State Combiner functionality to allow several WdgM instances, each running on a separate processor core, to share and trigger a single watchdog device. The WdgIf was developed according to AUTOSAR version 4.0.1 [1].

The WdgIf is compatible with this AUTOSAR version, but not fully compliant. For the deviations, see section Deviations. In any case, if the WdgIf is used with AUTOSAR 4.0.1 or another version, all requirements described in the Safety Manual [2] must be fulfilled.

This user manual does not cover safety-related topics. For safety-related requirements for the integration and the application of the WdgIf, refer to the Safety Manual [2].

2.1 Architecture Overview

The following figure shows where the WdgIf is located in the AUTOSAR architecture.

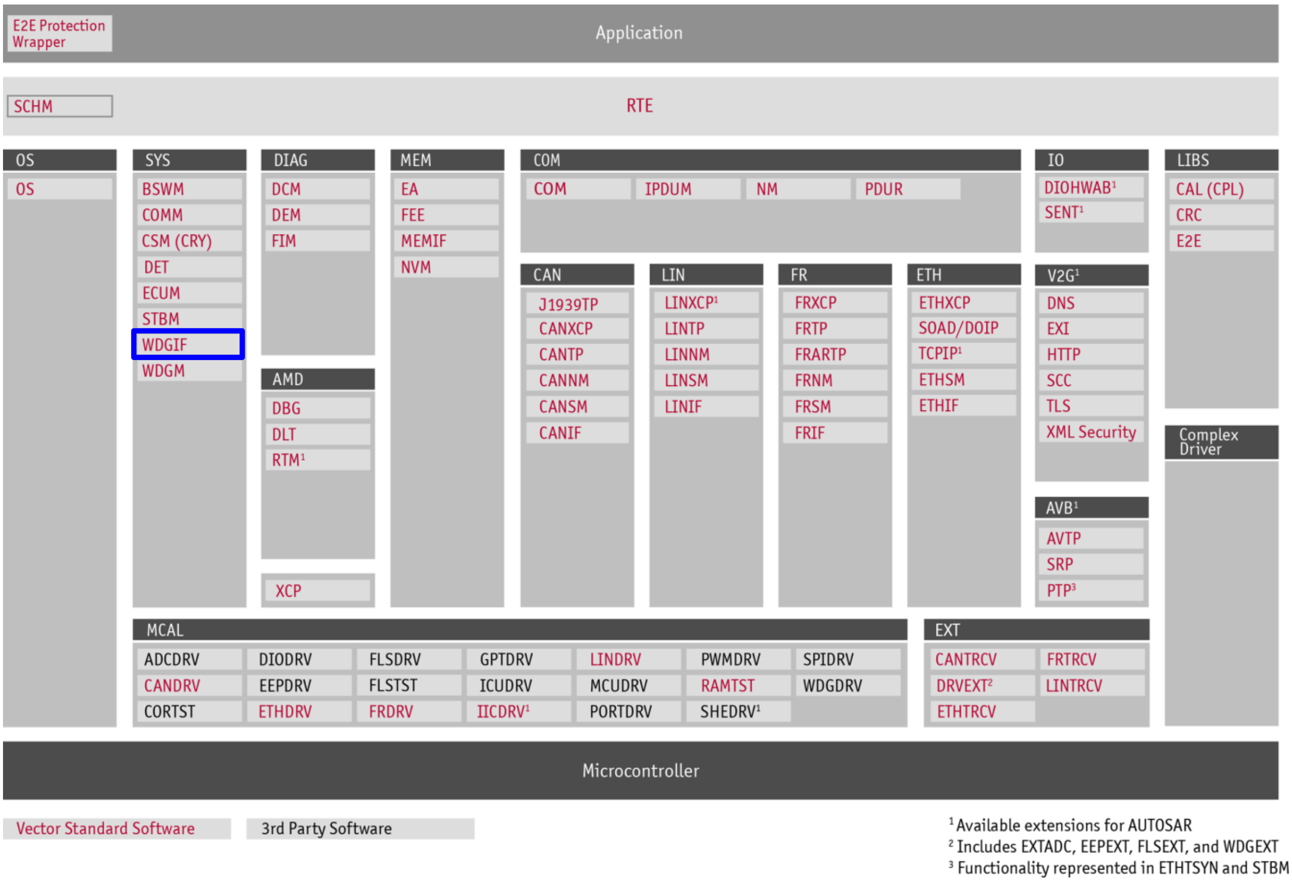

Figure 2-1 AUTOSAR 4.x Architecture Overview

The WdgM Stack consists of the hardware-independent modules Watchdog Manager and Watchdog Interface (blue rectangle) and a hardware-dependent module Watchdog driver. Figure 2-2 shows the WdgM Stack with its modules in an AUTOSAR environment.

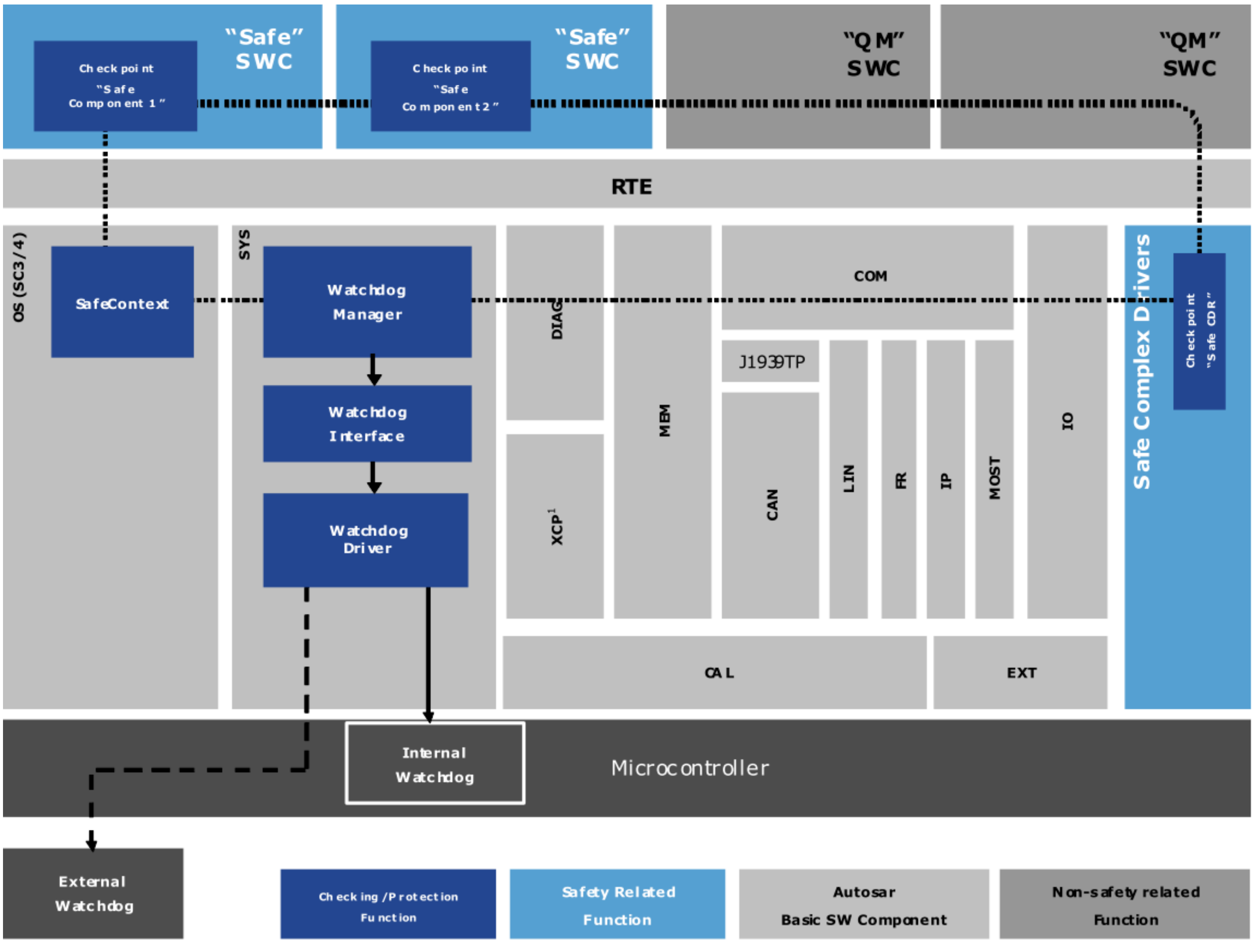

Figure 2-2 Watchdog Manager Stack in an AUTOSAR environment

The WdgM controls, through the WdgIf and the Wdg, the hardware-implemented watchdogs, which can be one or more internal or external watchdog devices.

Note
A watchdog device requires a hardware-dependent Wdg driver.

2.2 Basic Functionality of the WdgIf

The WdgIf is a platform-independent software module and provides an interface to one or more Watchdog driver modules for the WdgM. The WdgM addresses the watchdog devices through the WdgIf using a device index parameter (`DeviceIndex`). The `DeviceIndex` is used by the WdgIf to refer to a specific Wdg driver instance.

Figure 2-3 shows the layered structure of the Wdg Stack. The attached watchdog device can be internal or external.

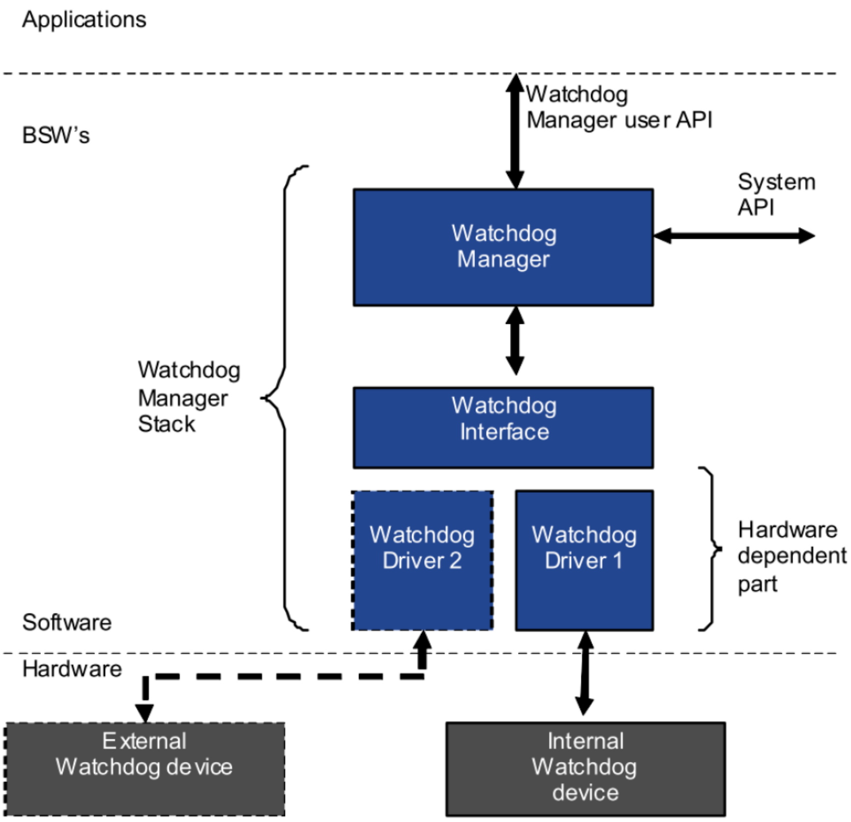

Figure 2-3 Layered structure of the Watchdog Interface

3 Functional Description

3.1 Features

The features listed in the following tables cover the complete functionality specified for the WdgIf.

The AUTOSAR standard functionality is specified in [1], the corresponding features are listed in the tables

> Table 3-1 Supported AUTOSAR standard conform features
> Table 3-2 Not supported AUTOSAR standard conform features

Vector Informatik provides further WdgIf functionality beyond the AUTOSAR standard. The corresponding features are listed in the table

> Table 3-3 Features provided beyond the AUTOSAR standard

The following features specified in [1] are supported:

Supported AUTOSAR Standard Conform Features
The WdgIf provides uniform access to services of the underlying watchdog drivers like mode switching and setting trigger conditions.

Table 3-1 Supported AUTOSAR standard conform features

3.1.1 Deviations

The following features specified in [1] are not supported:

Not Supported AUTOSAR Standard Conform Features
No deviations.

Table 3-2 Not supported AUTOSAR standard conform features

3.1.2 Additions/ Extensions

The following features are provided beyond the AUTOSAR standard:

Features Provided Beyond The AUTOSAR Standard
The WdgIf module checks for development errors independently from the configuration parameter `WdgIfDevErrorDetect` but reports to the AUTOSAR module Development Error Tracer (DET) only if `WdgIfDevErrorDetect` is set to true.
In case of multi-core systems, the WdgIf supports the State Combiner functionality which is not specified by AUTOSAR.
If the State Combiner functionality is used, then the WdgIf calls the functions `GetSpinlock()` / `ReleaseSpinlock()` (if configuration parameter `WdgIfStateCombinerUseOsSpinlock` is `true`) or the functions `Appl_GetSpinlock()` / `Appl_ReleaseSpinlock()` (if configuration parameter `WdgIfStateCombinerUseOsSpinlock` is `false`) in order to use spinlock functionality for inter-core synchronization. For details, see section Services used by WdgIf.

Table 3-3 Features provided beyond the AUTOSAR standard

3.2 Operation in Multi-Core Systems

The WdgIf can also be integrated into **multi-core** systems. During the configuration of the WdgIf on several cores, it is important to consider how to connect each WdgM instance running on a processor core to the correct Wdg driver module or modules via the WdgIf. There are two possible approaches for configuring the WdgIf for a multi-core system:

- Independent watchdog devices

  Configuring the WdgIf module so, that the WdgM instances running on different processor cores trigger its own watchdog device independently from the other cores. An example of such a system is a multi-core processor which has one internal watchdog device for each core. A fault on a certain core results in a watchdog reaction from the core's own watchdog device. Depending on its setup this might be a processor reset or only a single core reset.

- WdgIf with a State Combiner

  Configuring the WdgIf module with a State Combiner so that the WdgM instances running on different processor cores can share one watchdog device and use it to cause a reset in case of an irreparable error. The watchdog device will be triggered only if no WdgM instance reports any error.

  An example is a multi-core processor with an external watchdog connected to it. A fault on any processor core results in a watchdog reset.

Note
A combination of the two approaches above is also possible.

3.2.1 Independent Watchdog Devices

The WdgIf is configured to enable each Watchdog stack instance running on a separate processor core to trigger its own watchdog device independently from the Watchdog stack instances running on the other cores. Whether the watchdog device causes a processor reset or a core reset depends on the device's configuration. In this case, the Watchdog stack instance running on each processor core is acting as if it is running on an independent single-core system. Configuring this scenario is also very similar to the single-core configuration. However, it needs to be ensured that the watchdog device for a certain core is connected to the correct WdgM instance. Furthermore, the configuration parameter `WdgIfUseStateCombiner` must be set to `false`.

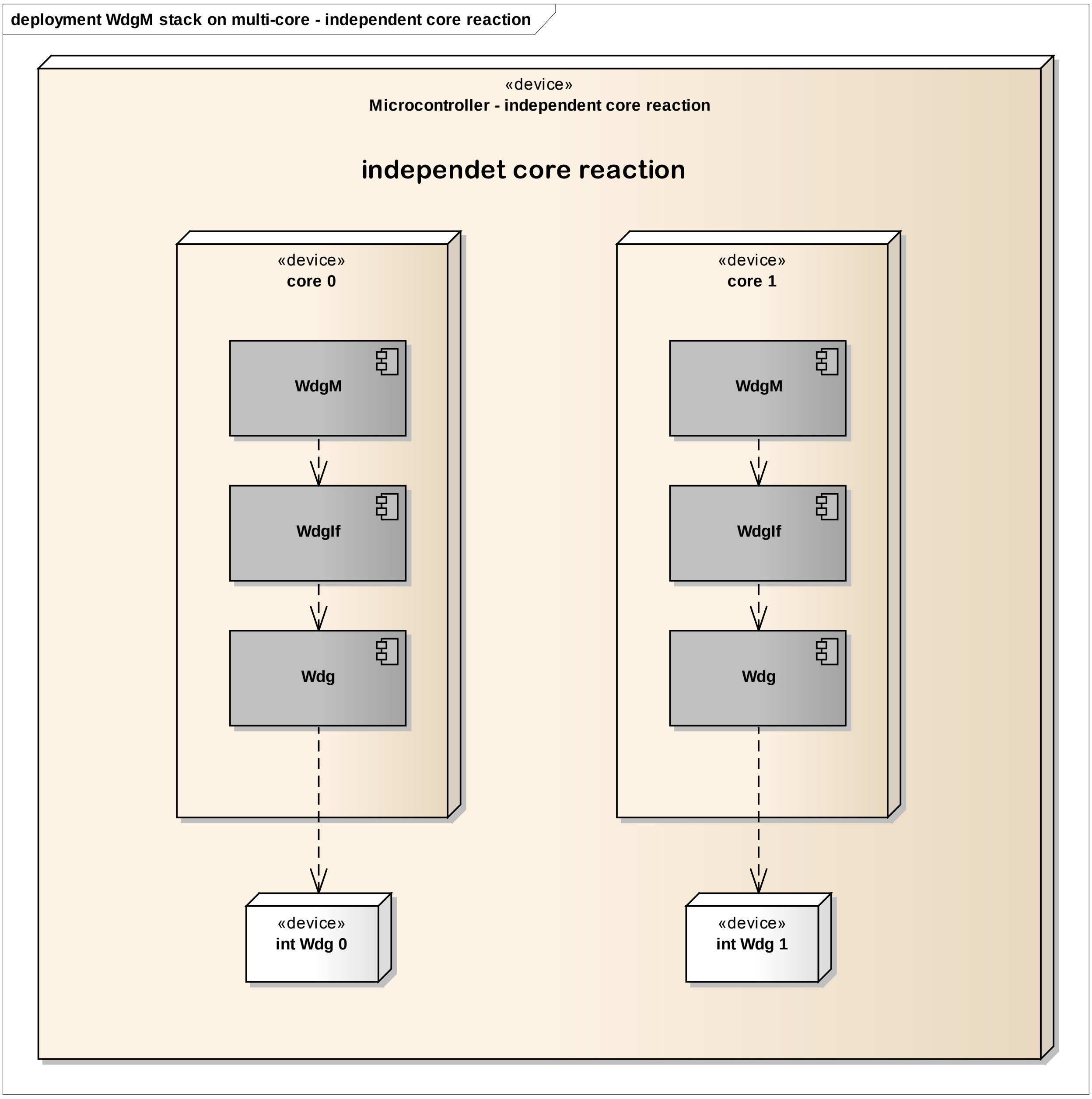

Figure 3-1 WdgM Stack on a multi-core system using WdgIf to address independent watchdogs for each core

3.2.2 WdgIf with a State Combiner

The **State Combiner** is a **platform-independent** piece of software that is implemented as an optional feature of the WdgIf module. Its purpose is to enable WdgM instances running on different processor cores to share one watchdog device. The State Combiner acts as following:

> If an error during the WdgM supervision is detected on a core, then the WdgM instance on this core requests a reset, which the State Combiner retransmits to the watchdog device.

> Furthermore, the State Combiner monitors the trigger pattern of the WdgM instances in order to detect runtime errors such as trigger omissions (e.g. one of the processor cores stopped working) or too frequent triggers (e.g. due to scheduling problems, an WdgM instance is invoked too frequently).

> The State Combiner triggers the watchdog device only if none of the WdgM instances requests a reset and the trigger patterns of all WdgM instances are correct.

> The State Combiner feature can be enabled by setting the configuration parameter `WdgIfUseStateCombiner` to `true`.

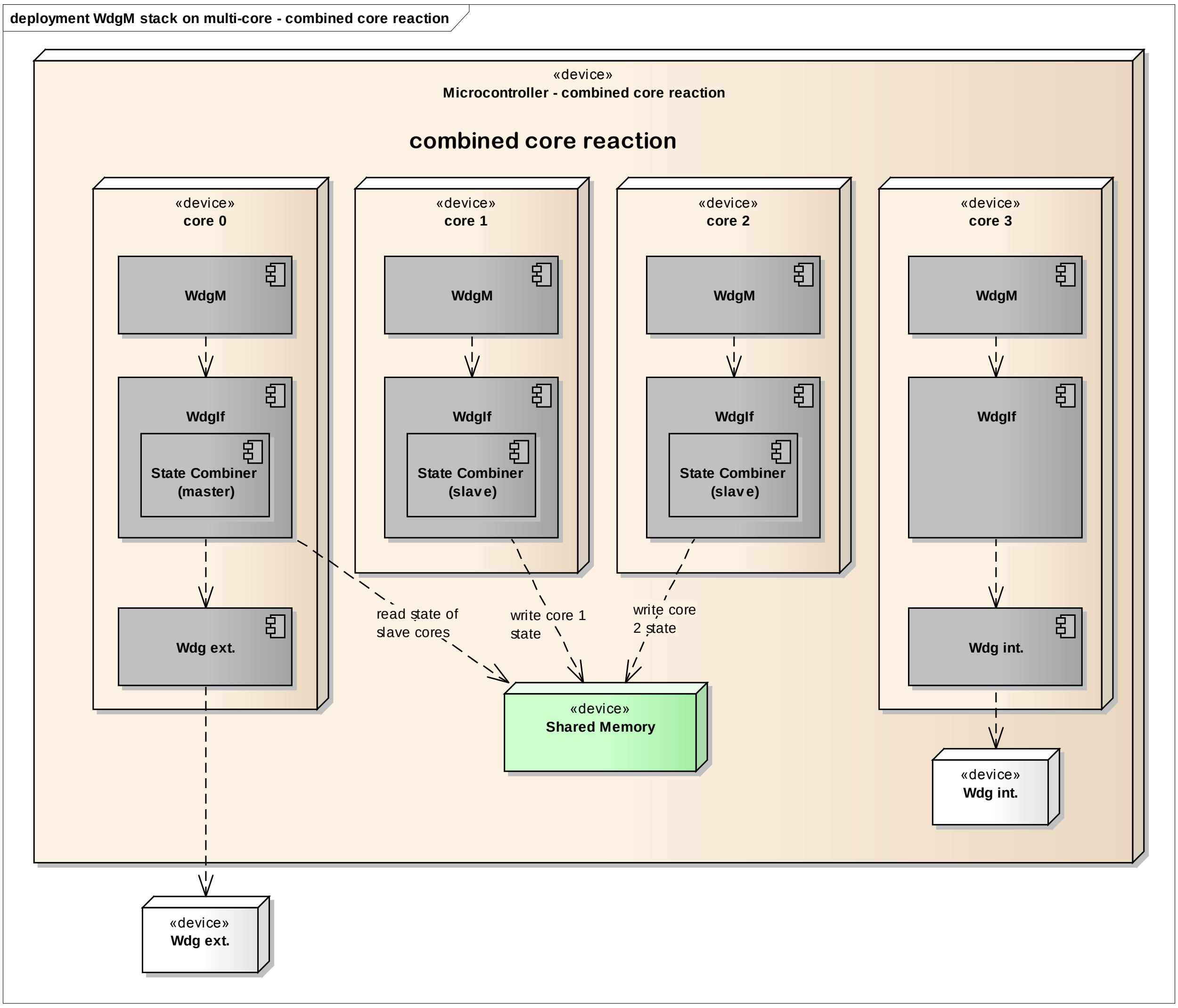

Figure 3-2 WdgM Stack on a multi-core system using the State Combiner for a combined core reaction

If enabled, the State Combiner instance on **one processor core** is configured to work in **master mode**, which triggers the actual watchdog device, while State Combiner instances on the **other processor** cores are configured to work in **slave mode**. In the following the State Combiner instance configured to work in master mode is referred to as **master** and the State Combiner instance(s) configured to work in slave mode as **slave(s)**. The slaves do not trigger a watchdog device but only communicate with the master via shared memory. The master triggers the actual watchdog device if the global status of the WdgM instances on all cores is other than `STOPPED`. Therefore, as soon as the WdgM instance on at least one core has reached the global status `STOPPED` (i.e. an irreparable error was detected), the watchdog device is – depending on the configuration – reset or not triggered anymore.

Note
The State Combiner is not visible to the upper layer, i.e. the WdgM instances on each processor core.

The trigger process in case of a State Combiner is as follows:

> The WdgM instance on a processor core sends a trigger request to its underlying WdgIf instance. No watchdog device is triggered, but the corresponding State Combiner instance is invoked - either the master or a slave.

> The slave does not trigger but rather signals to the master the trigger request from the upper layer.

> If the slave detects an error, it will send a reset request to the State Combiner.

> Based on the trigger pattern of the slave (the sequence of the slave's trigger request signals over a certain period of time), the master evaluates whether the slave is running correctly.

> The master triggers the actual watchdog device if:

    > the master's overlying WdgM instance requested a valid watchdog trigger,

    > no slave requested a reset (no error reported by the slave's overlying WdgM instance), and

    > the trigger pattern of each slave is correct (based on the configuration).

The following must be configured so that the State Combiner is used by the overlying WdgM instances to trigger a single watchdog device for all processor cores:

> The WdgM instance running on the processor core that controls the physical watchdog device must be configured to send a trigger request to the master. (In the WdgIf's ECU configuration, the `WdgIfDeviceRef` parameter must be linked to a `WdgIfStateCombinerMaster` container of WdgIf instead of a `WdgIfDevice` container.) The trigger condition value of the WdgM needs to be set up according to the actual watchdog device.

> The WdgM instances running on the other processor cores must be configured to send a trigger request to a slave. (In the WdgIf's ECU configuration, the `WdgIfDeviceRef`

parameter must be linked to a `WdgIfStateCombinerSlave` container of WdgIf instead of a `WdgIfDevice` container.)

Note
The trigger condition value for a slave can be configured arbitrary. The only requirement is not configuring the value with 0.

> The master must be configured to trigger the watchdog device. (In the WdgIf's ECU configuration the parameter `WdgIfStateCombinerMasterWdgRef` must reference the watchdog device's driver.) The trigger condition value with which the driver will be triggered is given by the overlying WdgM and retransmitted to the watchdog device by the master.

> Following this configuration, the master checks the trigger requests of each slave and triggers the watchdog device only if each slave triggers correctly, no slave explicitly requested a reset, and the master was triggered correctly.

> A reset occurs in the following cases:

  > The WdgM instance triggering the master requests a reset – the reset request is immediately retransmitted to the watchdog device.

  > The WdgM instance triggering a slave requests a reset – the reset request is retransmitted to the watchdog device with the next invocation of the master.

  > The master detects a shared memory corruption – it checks the shared memory each time it is invoked – then the master immediately sends a reset request to the watchdog device.

3.2.2.1 Checking the Slave Trigger Pattern

Checking the trigger pattern of the slaves by the master is based on slave trigger counters which are stored in shared memory. Each counter contains the number of triggers for a specific slave. The slave increases its trigger counter each time it is being invoked with a valid trigger request by its overlying WdgM instance. The master checks the slave trigger counter once per master period or once per a multiple of the **master period**. This multiplicity factor is called **reference cycle** and the duration of time in which the master checks a slave once is called **check interval**. E.g., if the master checks a slave each time the master is invoked, then the reference cycle is 1 and the check interval is one master period; if the master checks the slave every other time the master is invoked, then the reference cycle is 2 and the check interval is 2 times the master period.

The master expects that the slave increases its trigger counter in every check interval by a certain number. This number depends on the master period, the slave period and their ratio to one another. The increase of the slave trigger counter must be at least 1. Otherwise the error case of a total slave outage cannot be detected.

Note
The reference cycle as well as the number of expected slave triggers might be different for each slave.

3.2.2.2 Operation of the State Combiner

There are two possible operation modes – synchronous or asynchronous mode. In the synchronous mode a check interval exists such that the number of slave invocations in one check interval is always constant. Therefore the master can be configured to expect a constant number of slave trigger counter increments. In the asynchronous mode no such constant check interval exists and the number of slave invocations in one check interval is variable. Therefore the master can only expect that the number of slave counter increments lies within a configured interval.

3.2.2.2.1 Synchronous Mode

Synchronous mode is given if a check interval can be chosen in which the number of slave triggers is always constant. This is the case if both following conditions apply:

> *No drifting*. The master and slave invocations do not drift apart. The ratio between master and slave period remains constant.

> *Sufficient invocation offset*. The slave invocation is done with a sufficient offset from the master invocation so that their invocation order is not affected by jitter (jitter effects are avoided).

  The jitter effects can be avoided if the offset between master and slave invocations is greater than the sum of the maximum possible jitter of the master invocation (j_m) and the maximum possible jitter of the slave invocation (j_s). Note that these are the jitters of the respective WdgM main functions invoking master and slave. Two offsets need to be considered:

  > The offset from the master invocation in which the master checks the slave to the next slave invocation must be greater than jm + js.

  > The offset from the slave invocation to the next master invocation in which the master checks this slave must be greater than jm + js as well.

The **benefit** of the synchronous mode is the shorter interval in which the master can check the number of slave triggers (leading to a shorter reaction time) as well as the guaranteed detection of all slave trigger errors. Furthermore, if the jitter becomes bigger than the configured offset, this will be detected as an error.

The **drawback** of the synchronous mode is that if the timing of the system must be changed during runtime (e.g. low power mode), then the ratio between master and slave invocation period must remain the same.

Following scenarios illustrate typical examples of the synchronous mode.

Figure 3-3 depicts an example of a scenario where master and slave have the same period (P_m = P_s). The master checks the slave once in each master period (reference cycle is 1) and it expects exactly one slave triggering. The offset is sufficient to avoid jitter effects.

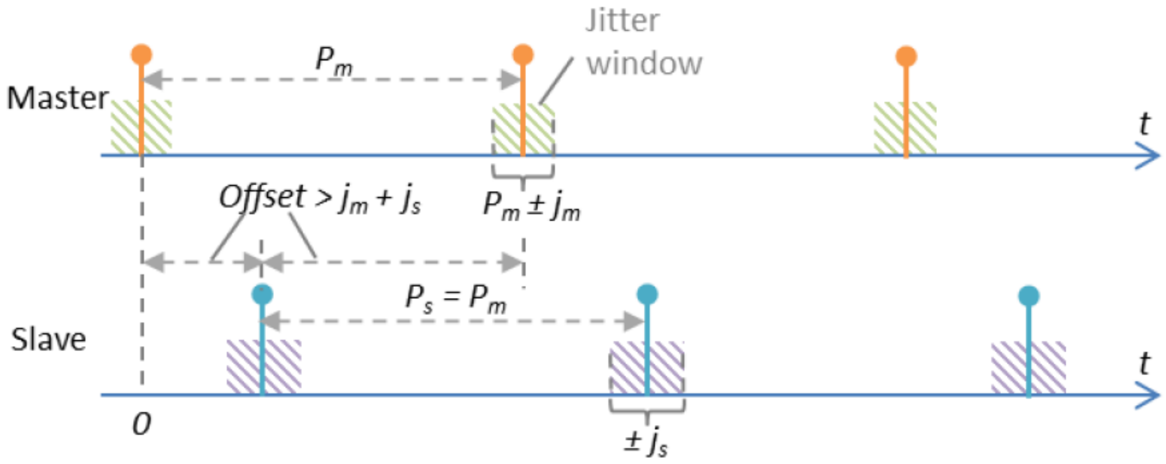

Figure 3-3 Master and slave run synchronously with a sufficient offset to avoid jitter effects (example 1)

Figure 3-4 shows an example of a scenario where the slave's period is a multiple of the master's period (in the example P_s = 2*P_m). As a consequence, the number of slave triggers within the check interval (reference cycle is 2) is always constant – one in this example. The offset is sufficient to avoid jitter effects.

Note
When master and slave periods are referred in this text, the configured periods are meant. Due to jitter, the actual periods might, of course, be slightly different. However, it is important that the conditions for synchronous mode apply.

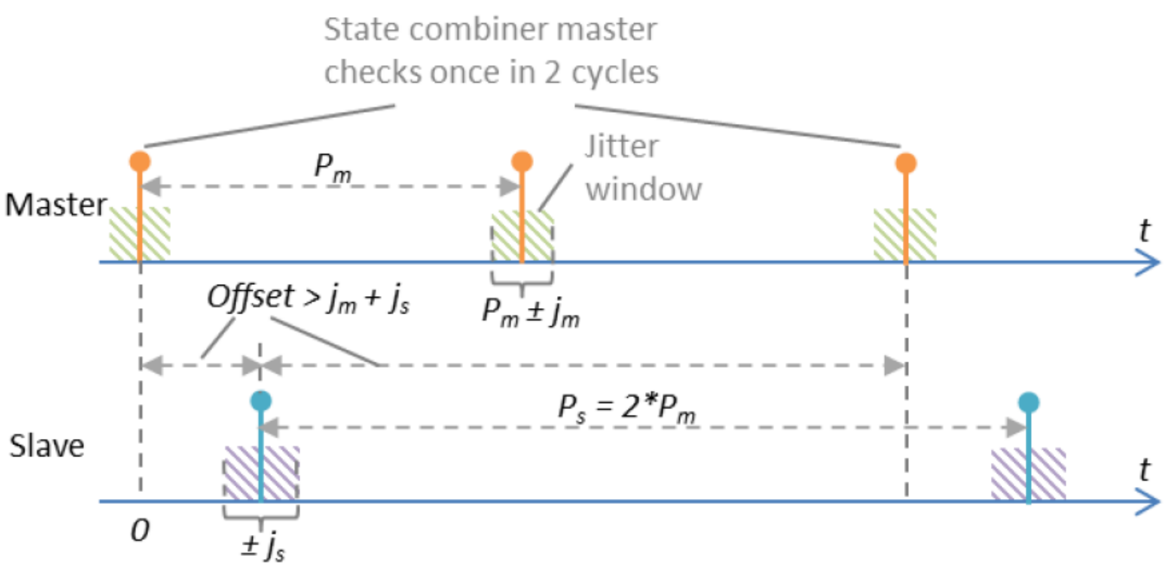

Figure 3-4 Master and slave run synchronously with a sufficient offset (example 2)

Figure 3-5 shows an example of a scenario where the master's period is a multiple of the slave's period (in the example $P_m = 2*P_s$). Again, the number of slave triggers within a master's check interval (reference cycle is 1) is always constant – two in this example.

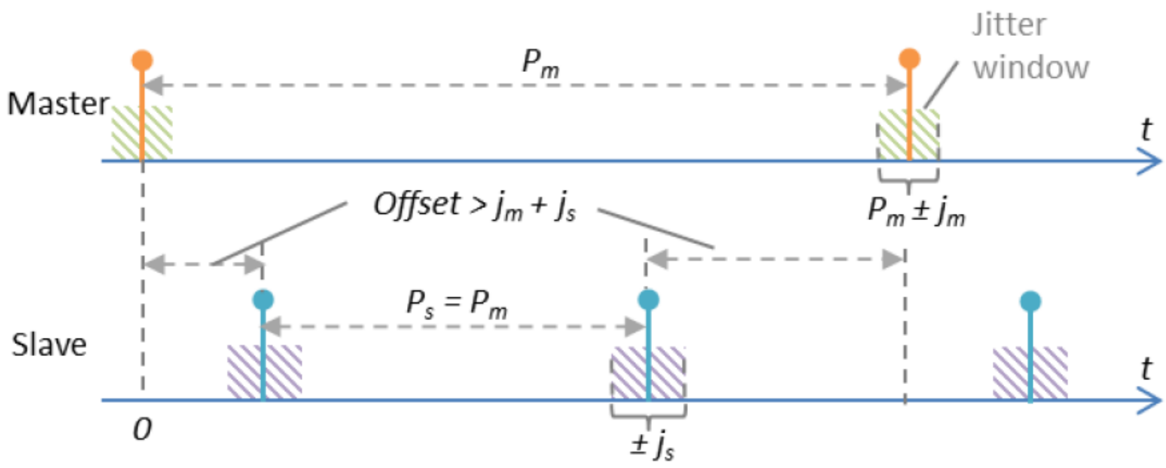

Figure 3-5 Master and slave run synchronously with a sufficient offset (example 3)

The Synchronous Mode is strongly recommended, because it results in the most accurate slave monitoring that can be reached with a software State Combiner as well as in the shortest worst case reaction time in case of slave trigger errors. Furthermore, it detects every kind of trigger error because the exact number of expected triggers is known.

3.2.2.2.2 Asynchronous Mode

Asynchronous mode is given if the synchronous mode cannot be applied – in asynchronous mode no check interval can be chosen such that the number of slave triggers is constant in each check interval. This is the case if at least one of the following applies:

> *Drifting*. Master and Slave invocations drift from one another.

> *Insufficient invocation offset resulting in jitter effects*. The offset between master and slave invocations is such that the jitter effects result in a variable invocation pattern (number of slave triggers changes between check intervals).

As a consequence, the master can only check whether the actual number of slave triggers is within a certain interval.

The **benefit** of the asynchronous mode is that if the timing of the system must be changed during runtime, then the ratio between master and slave invocation period need not remain the same. In this case, the State Combiner is usually configured to compute the expected number of slave triggers dynamically.

The **drawback** of the asynchronous mode is the necessity of introducing a tolerance when checking the slaves – the number of expected slave triggers lies within an interval. This results in a greater reference cycle and in potentially overlooking slave trigger errors.

Simple scenarios for each of the two reasons that lead to asynchronous mode are discussed below. After that, one examples illustrating the drawback of the asynchronous mode – the potential overlooking of trigger errors – are presented.

Scenario 1: Asynchronous Mode due to Drifting

Master and slave invocations drift from each other.

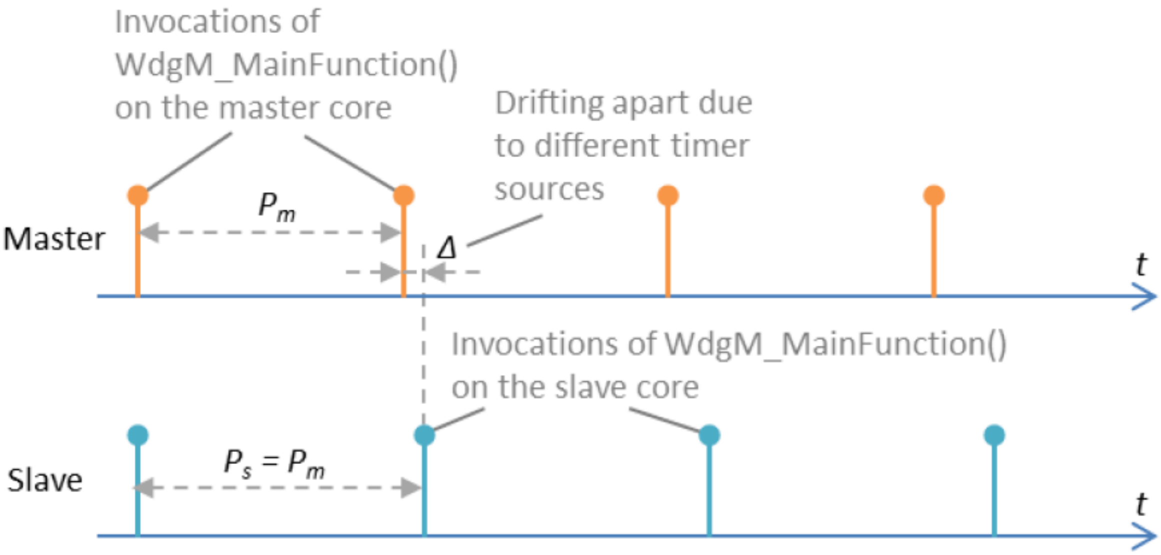

Figure 3-6 Master and slave drifting apart although they have the same configured period ($P_m = P_s$)

In this example, the master period and the slave period have the same configured length but their clocks drift with some rate Δ (positive or negative). The master must check once in n master periods whether the number of slave triggers is within an interval `[tr1; tr2]`.

Note
The exact reference cycle n and the interval of the number of expected slave triggers depend on the master and slave periods. With increasing jitter the reference cycle also increases.

Scenario 2: Asynchronous Mode due to Insufficient Offset (Jitter)

Master and slave do not drift apart. But they are invoked at the same points of time or close enough to one another so that the jitter affects their sequence. This is illustrated in Figure 3-7.

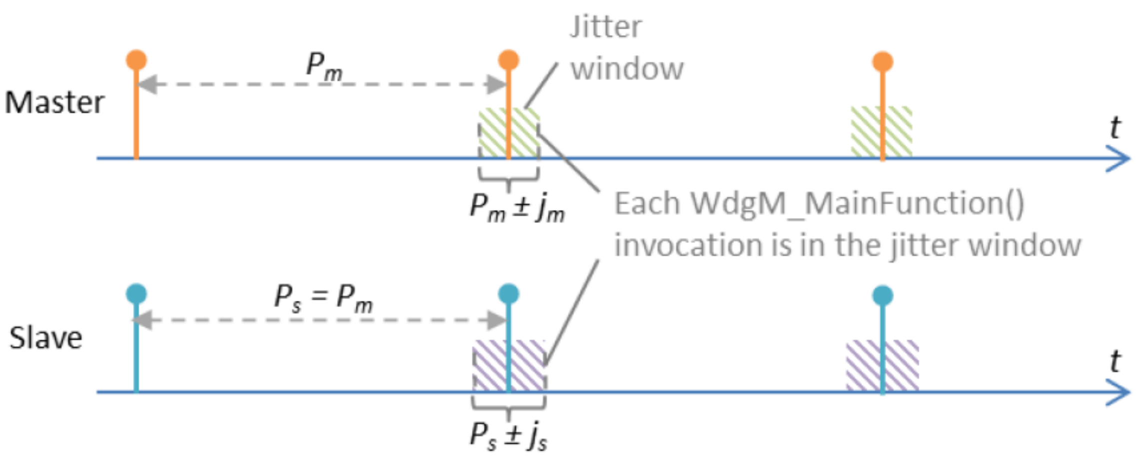

Figure 3-7 Master and slave do not drift from each other but jitter effects occur

In this case, the master and slave are running synchronously, but due to the jitter and the insufficient offset between master and slave invocations the trigger pattern is unpredictable. For the master and a slave running with the same period the same values are derived as for the asynchronous scenario with drifting above – the master checks the slave once in every second master period (reference cycle is 2) and the number of expected slave triggers lies in the interval between 1 and 3 inclusively.

Example of Overlooking Trigger Errors: Slave Trigger Omissions

Figure 3-8 shows an example of how a trigger omission can be overlooked by the master. Let the expected slave trigger counter interval be `[1; 2]`. During the first check interval, the slave is invoked correctly (as expected by the master). During the second check interval, the slave should have triggered two times, but one trigger is omitted – the master cannot detect this trigger error, since the trigger counter interval is not violated. The third check interval shows zero triggers and this is out of the interval, hence the trigger error is detected.

Note
In this example, a minimum of two consecutive slave invocation omissions will always be detected by the master.

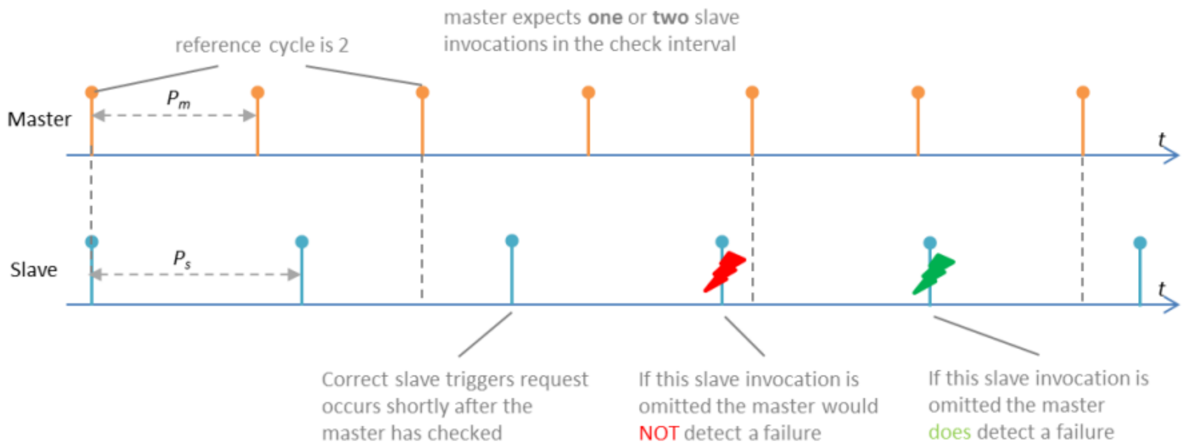

Figure 3-8 Slave skipping one trigger is not necessarily detected by master in asynchronous mode

Note
Due to the drawbacks, using the asynchronous mode should be avoided and, if possible, the synchronous mode should be used!

3.2.2.3 Worst Case Delay

The **delay** of the State Combiner is defined as the duration from the point in time when a failure occurs on the slave and the point in time when this failure is escalated to the watchdog device by the master. The failure on the slave can be a failure detected by the WdgM running on the slave's core or a failure which results in erroneous triggering of the slave. Here, a failure on the slave is a slave trigger outage, i.e. discontinuation of the slave triggers, and the worst case delay refers to this slave trigger error only.

Note
Drifting of the slave triggering might lead to a longer detection time (in both, synchronous and asynchronous mode) or might be overlooked by the master (in asynchronous mode only). Occasional slave trigger omissions might be overlooked by the master only in asynchronous mode, but they are detected in synchronous mode.

Note
Reset requests from the slave are detected by the master at the end of the current master period (and not at the end of the current check interval) in both, synchronous and asynchronous mode.

The upper limit for the **worst case delay of the State Combiner (WCD)** in synchronous mode is the double maximum duration of the check interval: `WCD < 2*n*`T_m, where T_m is the WdgM configuration parameter `WdgMTriggerWindowCondition` set on the master core and `n` is the **reference cycle** with which the master checks the slave.

Note
T_m is the worst case actual period of invocation of the master's `WdgM_MainFunction()`, and it is limited by the watchdog device. T_m can also be expressed as the configured master invocation period plus the maximum possible jitter of this invocation: $T_m = P_m + j_m$.

The **worst case scenario** happens under the following conditions (illustrated in Figure 3-9). The slave is triggered shortly after the master has successfully checked the slave triggers. However, the slave fails right afterwards and is not being triggered anymore, it is not able to directly inform the master of a failure either. At the end of the current check interval the master still evaluates the slave as OK if the number of slave triggers is within the expected interval despite the trigger error. Yet, the next time the master core checks the slave core, it detects that the slave has stopped triggering (at the end of the third check interval shown in the figure).

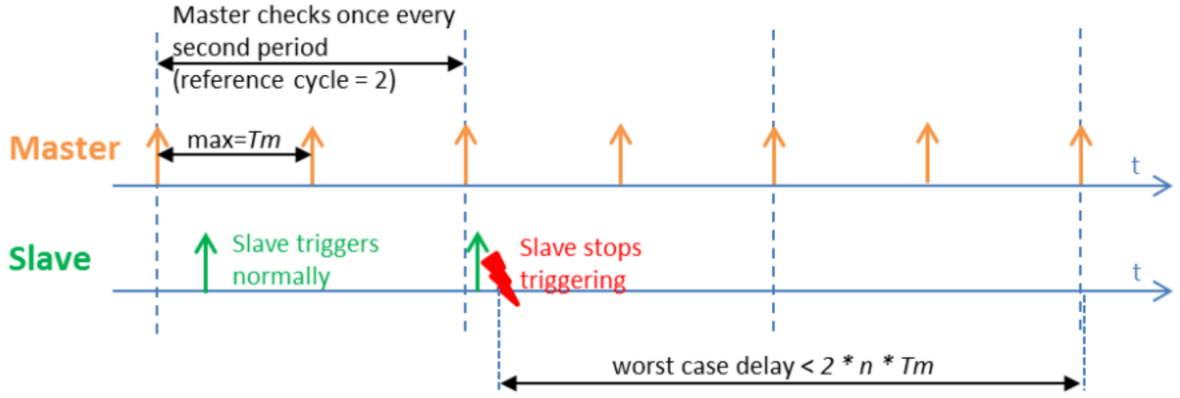

Figure 3-9 Worst case delay of the State Combiner

Note
Slave trigger errors that do not lead to violation of the expected number of slave triggers interval cannot be detected by the master!

3.2.2.4 Worst Case Evaluations

The WdgIf Fault Reaction Time does not depend on the monitoring feature, but on the following three aspects:

> whether a State Combiner is used or not,

> whether an immediate reset or discontinuing of triggers is configured,

> whether the fault is detected in the master application SW or a slave application SW (if a State Combiner is used).

There exist 6 different combinations of the three aspects listed above:

Case	State Combiner used	Escalation kind	Fault occurs in
1	Yes	Immediate Reset	Master SW application
2	Yes	Immediate Reset	Slave SW application
3	Yes	Discontinuing of Triggers	Master SW application
4	Yes	Discontinuing of Triggers	Slave SW application
5	No	Immediate Reset	n/a
6	No	Discontinuing of Triggers	n/a

Table 3-4 Combinations for worst case evaluation

The WdgIf Fault Reaction Time of every combination is discussed in the following:

Case 1 - State Combiner, immediate reset, fault in master, **Case 5** - No State Combiner, immediate reset:

The WdgIf escalates the reset request immediately to the Wdg device. The WdgIf Fault Reaction Time for case 1 and case 5 is always 0 (in any case, there is no more cycle consumed - not counting the code execution).

Case 2 - State Combiner, immediate reset, fault in slave:

> The slave writes an immediate reset request to the shared memory of the State Combiner.

> The master reads the request at the next call of `WdgM_MainFunction()` and initiates the immediate reset.

The worst case happens

> when the master calls its `WdgM_MainFunction()`,

> the slave writes the reset request immediately afterwards and

> the master calls its `WdgM_MainFunction()` with max. possible delay (`WdgMTriggerConditionValue`(master)).
> As Figure 3-10 shows, the WdgIf Fault Reaction Time is `WdgMTriggerConditionValue`(master).

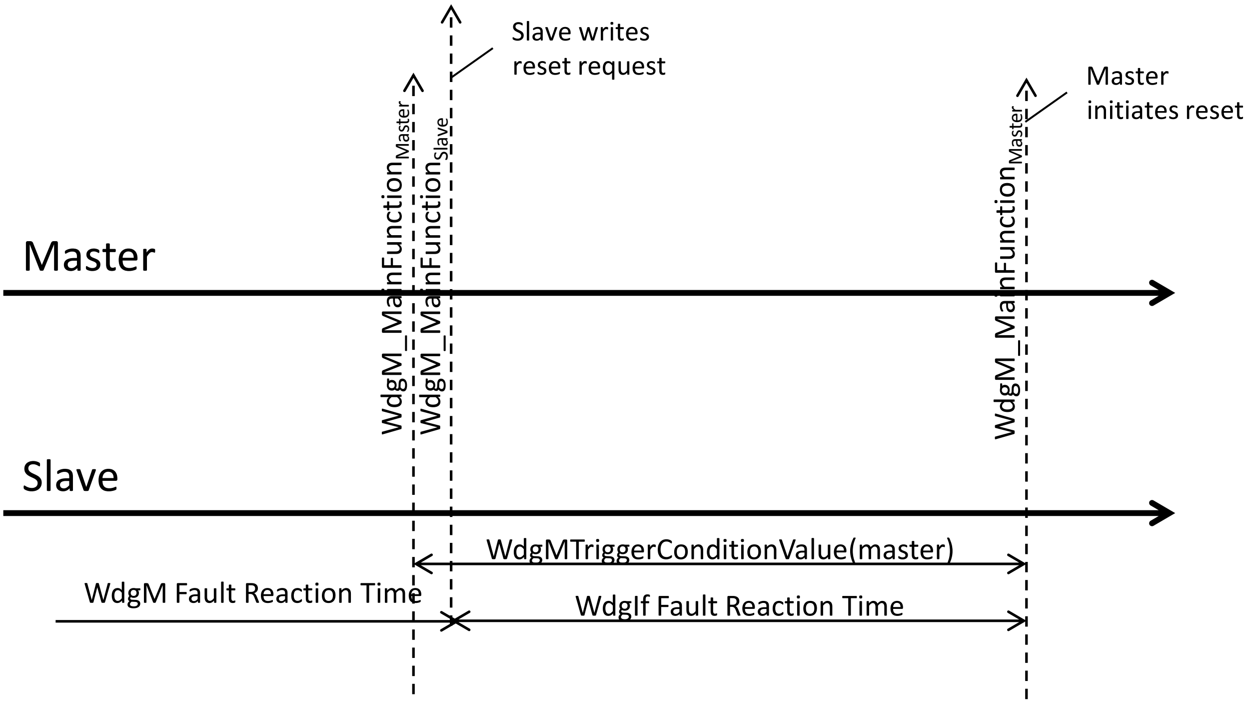

Figure 3-10 Worst case evaluation Case 2

Case 3 - State Combiner, discontinuing of triggers, fault in master, **Case 6** - No State Combiner, discontinuing of triggers:

There is no action or delay on the WdgIf level. The WdgIf Fault Reaction Time for case 3 and case 6 is always 0 (in any case, there is no more cycle consumed - not counting the code execution).

Case 4 - State Combiner, discontinuing of triggers, fault in slave:

> The slave discontinues triggering.

> With every call of WdgM_MainFunction() on master side, the master checks how often the slave has triggered since the previous check.

> As soon as the number of slave triggers is outside the expected range, the master initiates an immediate reset. (This is not necessarily with the next call of WdgM_MainFunction() on master side.)

The worst case happens when

> the master checks the number of triggers on slave side since the previous check,

> the slave sends an allowed number of triggers (with respect to the next check on master side) immediately afterwards,

> the WdgM Fault Reaction Time ends and the slave discontinues triggering immediately afterwards.

Note
Then the WdgIf Fault Reaction Time is (almost):

2 * WdgIfStateCombinerReferenceCycle * WdgMTriggerConditionValue$_{Master}$,

where WdgIfStateCombinerReferenceCycle is the number of WdgMSupervisionCycle on master side between two checks of slave triggers.

Figure 3-11 demonstrates this:

> WdgIfStateCombinerReferenceCycle is 2,

> the slave sends an allowed number of triggers for the 1st check interval (i.e. one trigger) before the end of the WdgIf Fault Reaction Time,

> the master checks the slave triggers every 2nd call of WdgM_MainFunction (every 2nd WdgMTriggerConditionValue (T_M)),

> the discontinuing of slave triggers is detected at the end of the 2nd check interval.

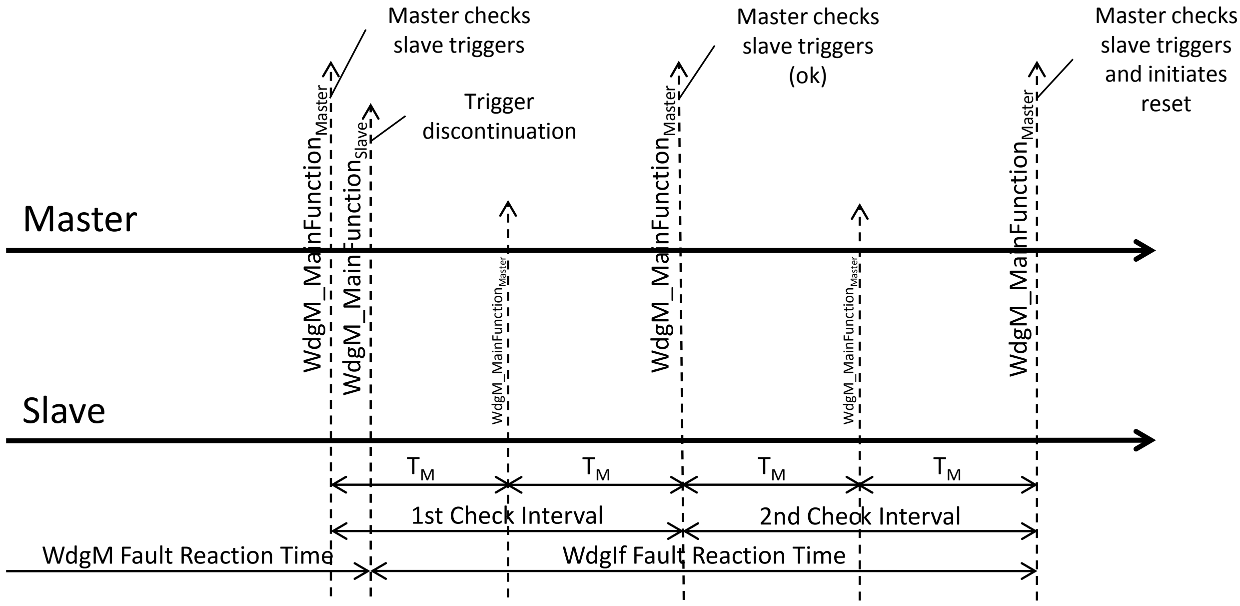

Figure 3-11 Worst case evaluation Case 4

3.2.2.5 Optimal Timing

The optimal timing results in minimal worst case delay. It can be reached when the reference cycle is minimal – which is 1. This applies for both, synchronous and asynchronous mode.

Following must apply so that the optimal reference cycle of 1 can be reached in **synchronous mode**. The period of the WdgM main function invoking the master (P_m) is a multiple of the period of the WdgM main function invoking the slave (P_s). If $P_m = n * P_s$, where `n = 1, 2, 3,…`, then the master can check the slave in each master period.

> Example (synchronous mode):
  > Master: P_m = 20ms
  > Slave: P_s = 10ms

Within one cycle of the master exactly 2 triggers of the slave are expected.

The worst case delay WCD to a failure in the slave is `40 ms`.

The following must apply so that the optimal reference cycle of 1 can be reached in **asynchronous mode**. The master period must be longer than the slave period

Example (asynchronous mode):

> Master: P_m = 21ms
> Slave: P_m 18ms

Within `n = 1` cycles of the master (at most `21 ms`) are 1 to 2 ticks of the slave expected.

The WCD for a failure in the slave is $2 * n * T_m = 42$ ms.

Note
Even with the optimal ratio between periods the drawbacks of the asynchronous mode described in chapter Asynchronous Mode apply.

3.2.2.6 Start-up Phase

If the slave starts together with or after the master, then the parameter `WdgIfStateCombinerStartUpSyncCycles` shall be set to some positive value n so that the master starts evaluating the slave triggering not from the first time the master is invoked after start up, but after the first n master periods.

> **Note**
> `n` must be big enough so that the master starts evaluating the slaves as soon as possible after the slaves started; and small enough so that the master does not start to evaluate before the slaves started.

A typical start-up phase setup is illustrated in Figure 3-12:

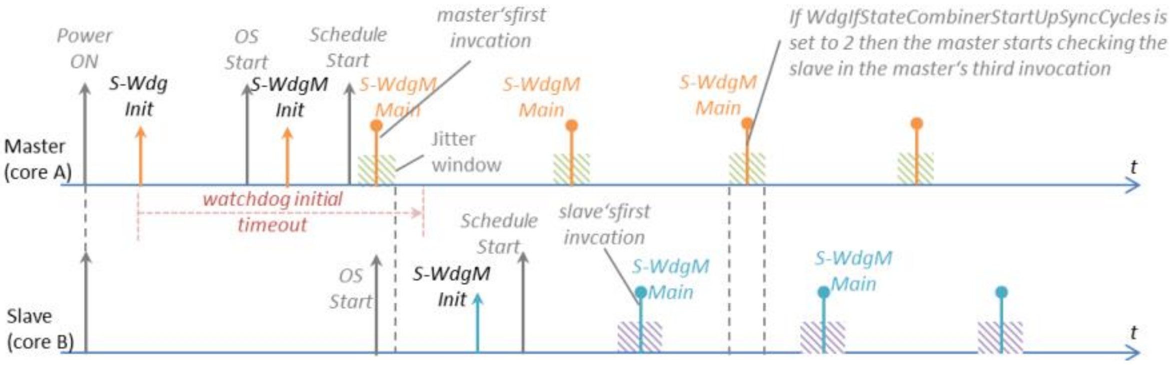

Figure 3-12 Start-up phase, master starts before slave

The slave (running on some processor core B) starts later than the master (running on processor core A). The `WdgIfStateCombinerStartUpSyncCycles` parameter is set to 2 so that the master starts checking the slave after the slave has started. Before the slave starts, the master triggers the watchdog device only according to the trigger requests of the master's overlying main function. Note, however, that if a slave's main function detects a failure and explicitly requests a reset, then the master reacts even during the start-up phase and retransmits the reset request to the watchdog device.

3.2.2.7 Changing the Monitoring Period During Runtime

Changing the monitoring period means that either the processor frequency or the period of invocation of master or slave is changed.

3.2.2.7.1 Changing the Monitoring Period in Synchronous Mode

If the monitoring period in a synchronous mode needs to be changed, several things need to be considered. The number of slave triggers within one check interval must remain the same and

- the change of the monitoring period must be made simultaneously on master and slave.

It is recommended that such a monitoring period change is not made while any instance of the WdgM Stack is being executed.

Figure 3-13 shows an example of monitoring period change in synchronous mode.

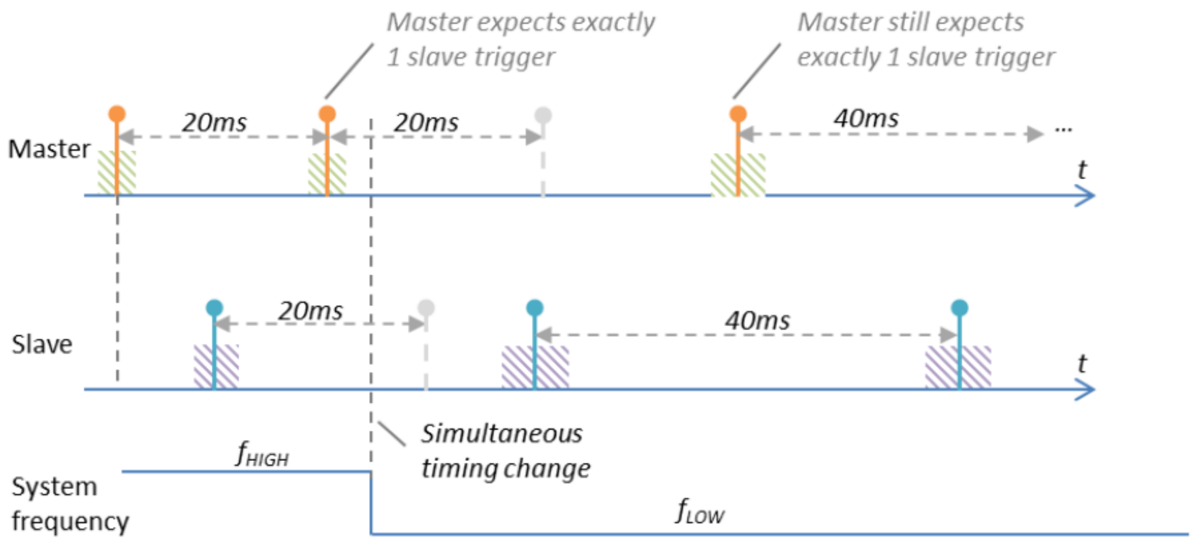

Figure 3-13 Start-up phase, master starts before slave

3.2.2.7.2 Changing the Monitoring Period in Asynchronous Mode

If the monitoring period in asynchronous mode needs to be changed, several things have to be considered.

If the State Combiner is configured in asynchronous mode, then for any change of the master period or slave period the following restriction applies:

> After the change the slave must not violate the interval of expected number of triggers.

In order to meet the previous restriction following recommendations apply:

> It is recommended that the ratio between master and slave period remains the same.

> It is recommended that the monitoring period change is done simultaneously for master and slave.

> It is recommended that such a monitoring period change is not made while any instance of the WdgM Stack is being executed.

3.2.2.8 Shared Memory

The State Combiner instances use shared memory to communicate. Every counter increment of every slave is written to this memory area. The master reads out the shared memory in order to check the counter increments against the expected counter increments. The slave's trigger requests increment the respective slave's trigger counter in shared memory. A reset request from the slave is also stored in the shared memory to inform the master. All data in the shared memory is also stored with inverse value in order to ensure the detection of memory corruption.

Access to the shared memory is protected against concurrent access. The shared memory is only written by the slaves and only read by the master. This is achieved by a mutex/semaphore that is configured for this shared memory block.

3.2.2.9 Limitations of the State Combiner Implementation

The State Combiner layer has the following limitations:

> Only one watchdog device can be connected to the master and be triggered. Other watchdog devices can, however, be directly connected with any WdgIf instance (ECU description container `WdgIfDevice`) and not via State Combiner.

3.3 Memory Sections

3.3.1 Code and Constants

Following memory sections need to be set up for WdgIf's code:

Section	Description
WDGIF_START_SEC_CODE / WDGIF_STOP_SEC_CODE	Set up manually, e.g. in `MemMap.h`.

Table 3-5 Code and Constants

Following memory sections need to be set up for WdgIf's constants:

Section	Description
WDGIF_START_SEC_CONST_UNSPECIFIED / WDGIF_STOP_SEC_CONST_UNSPECIFIED	Set up manually, e.g. in `MemMap.h`.

Table 3-6 WdgIf constants

3.3.2 Module Variables

Following memory sections need to be set up for WdgIf's module variables if the State Combiner functionality is used (otherwise the WdgIf uses no global variables):

3.3.2.1 Module Variables with MICROSAR Os Gen6 / AUTOSAR Os version 4.0

Section	Description
WDGIF_START_SEC_VAR_8BIT / WDGIF_STOP_SEC_VAR_8BIT, WDGIF_START_SEC_VAR_16BIT / WDGIF_STOP_SEC_VAR_16BIT	If the configuration parameter `WdgIfGlobalMemoryAppTaskRef` is set, then these sections are renamed according to the configured OS application (the prefix "`WDGIF_`" is converted to "`<OSApp>_`", where `<OSApp>` is the name of the OS application) and generated as part of `WdgIf_MemMap.h`. Otherwise they need to be set up manually, e.g. in `MemMap.h`.
WDGIF_GLOBAL_SHARED_START_SEC_VAR_UNSPECIFIED / WDGIF_GLOBAL_SHARED_STOP_SEC_VAR_UNSPECIFIED	These sections are always assigned in the generated file `WdgIf_MemMap.h` to OS sections and renamed to: `GlobalShared_START_SEC_VAR_UNSPECIFIED` / `GlobalShared_STOP_SEC_VAR_UNSPECIFIED` If other assignment is required, then they need to be set up manually, e.g. in `MemMap.h`.

Table 3-7 Module variables with MICROSAR Os Gen6 / AUTOSAR Os version 4.0

3.3.2.2 Module Variables with MICROSAR Os Gen7 / AUTOSAR Os version 4.2

Section	Description
WDGIF_START_SEC_VAR_8BIT / WDGIF_STOP_SEC_VAR_8BIT, WDGIF_START_SEC_VAR_16BIT / WDGIF_STOP_SEC_VAR_16BIT	If the configuration parameter WdgIfGlobalMemoryAppTaskRef is set, then these sections are renamed according to the configured OS application (the prefix "WDGIF_START_SEC" is converted to "OS_START_SEC_<OSApp>" and "WDGIF_STOP_SEC" is converted to "OS_STOP_SEC_<OSApp> ", where <OSApp> is the name of the OS application) and generated as part of WdgIf_MemMap.h. Otherwise they need to be set up manually, e.g. in MemMap.h.
WDGIF_GLOBAL_SHARED_START_SEC_VAR_UNSPECIFIED / WDGIF_GLOBAL_SHARED_STOP_SEC_VAR_UNSPECIFIED	These sections are always assigned in the generated file WdgIf_MemMap.h to OS sections and renamed to: OS_START_SEC_GLOBALSHARED_VAR_UNSPECIFIED / OS_STOP_SEC_GLOBALSHARED_VAR_UNSPECIFIED If other assignment is required, then they need to be set up manually, e.g. in MemMap.h.

Table 3-8 Module variables MICROSAR Os Gen7 / AUTOSAR Os version 4.2

3.4 Error Handling

3.4.1 Development Error Reporting

By default, development errors are reported to the DET using the service `Det_ReportError()` as specified in [1], if development error reporting is enabled (i.e. pre-compile parameter `WdgIf_DEV_ERROR_DETECT==STD_ON`).

If another module is used for development error reporting, the function prototype for reporting the error can be configured by the integrator, but must have the same signature as the service `Det_ReportError()`.

The reported WdgIf ID is 43 (decimal).

The reported service IDs identify the services which are described in chapter 5.3. The following table presents the service IDs and the related services:

Service ID	Service
0x01u	WdgIf_SetMode
0x02u	WdgIf_SetTriggerCondition
0x03u	WdgIf_GetVersionInfo
0x04u	WdgIf_SetTriggerWindow

Table 3-9 Service IDs

The errors reported to DET are described in the following table:

Error Code	Description
0x01u	API service called with wrong device index parameter
0x02u	API service called with NULL_PTR as parameter

Table 3-10 Errors reported to DET

4 Integration

The delivery of the WdgIf contains the files which are described in the chapters 4.1.1 and 4.1.2:

4.1.1 Static Files

File Name	Description
WdgIf.c	WdgIf implementation
WdgIf.h	WdgIf API definitions and function declarations
WdgIf_Types.h	WdgIf type definitions
WdgIf_Cfg.h	Type definitions for the configuration data in generated files

Table 4-1 Static files

4.1.2 Dynamic Files

The dynamic files are generated by the WdgIf generator.

File Name	Description
WdgIf_Lcfg.c	Generated configuration of the component.
WdgIf_Lcfg.h	Generated header file for the configuration of the component.
WdgIf_Cfg_Features.h	This file contains all preprocessor options for the component.
WdgIf_MemMap.h	This file contains memory sections relevant for the State Combiner functionality.

Table 4-2 Generated files

5 API Description

This section describes the types, functions and interfaces that are imported or provided by the WdgIf software layer.

5.1 Type Definitions

This section describes the types of the parameters passed to the API functions of the WdgIf.

Type Name	C-Type	Description	Value Range
WdgIf_InterfaceFunctionsType	c-struct	Provides pointers to the platform-specific APIs.	`Std_ReturnType (*Wdg_SetMode_AR) (WdgIf_ModeType)`
			`void (*Wdg_SetTriggerCondition_AR) (uint16)`
WdgIf_InterfaceFunctionsPerWdgDeviceType	c-struct	Connects platform-dependent functions to a physical watchdog in order to allow several watchdogs of the same platform to work simultaneously (e.g., external watchdogs).	`const WdgIf_InterfaceFunctionsType* WdgFunctions` Pointers to the platform-specific watchdog driver functions. **Note:** If the State Combiner is enabled, the NULL pointer is set instead of a pointer to the driver functions.
			`uint8 WdgInstance` Index of the physical watchdog instance within this platform. Note: If the State Combiner is enabled, the parameter `WdgInstance` is used to address the State Combiner instance instead of a physical watchdog device. **Note:** This parameter is used only if the State Combiner is used (preprocessor switch `WDGIF_USE_STATECOMBINER` is `STD_ON`).
WdgIf_InterfaceType	c-struct	Main WdgIf configuration structure	`const uint8 NumOfWdgs` Number of watchdogs supported in the WdgIf
			`const WdgIf_InterfaceFunctionsPerWdgDeviceType* WdgFunctionsPerDevice`

			Reference to the watchdog driver functions and watchdog device instances
			`const WdgIf_StateCombinerConfigType *WdgIfStateCombinerConfig` Pointer to State Combiner common specific configuration data. Part of the structure only if State combiner is used (`WDGIF_USE_STATECOMBINER` is `STD_ON`).
WdgIf_ModeType	enum	Mode of the Watchdog	`WDGIF_OFF_MODE` Watchdog disabled
			`WDGIF_SLOW_MODE` Long timeout period (slow triggering)
			`WDGIF_FAST_MODE` Short timeout period (fast triggering)

Table 5-1 WdgIf Type Definitions

5.2 State Combiner Type Definitions

This section describes the State Combiner types in case the State Combiner functionality is enabled.

Type Name	C-Type	Description	Value Range
WdgIf_StateCombinerSharedMemory	c-struct	State Combiner global shared data. Read by the master and written by all slave devices. Contains the current `WindowStart` and `Timeout` values of the slave devices and the `Counter` values. This is an array with an element for each slave.	`uint16 SlaveCounterValue` Current slave's trigger counter value.
			`uint16 SlaveCounterValue_INV` Inverted value of the current `Timeout` of the slave's trigger request.
WdgIf_StateCombinerSlaveTriggerPatternType	c-struct	Configuration structure for configuring State Combiner. This is an array with an element for each slave.	`uint16 WdgIfStateCombinerReferenceCycle` Defines the reference cycle with which the master will check the slave.
			`uint16 WdgIfStateCombinerSlaveIncr`

			ementsMin Minimal number of expected slave triggers in one master check interval.
			uint16 WdgIfStateCombinerSlaveIncrementsMax Maximal number of expected slave triggers in one master check interval.
WdgIf_StateCombinerConfigType	c-struct	State Combiner configuration structure	uint8 WdgIfStateCombinerNumberOfSlaves Number of slaves configured for the State Combiner.
			WdgIf_StateCombinerSpinlockIdType WdgIfStateCombinerSpinlockId Spinlock ID for synchronizing the access to the shared memory section.
			uint16 WdgIfStateCombinerStartUpSyncCycles Number of master cycles during start-up in which the master does not check the slave triggers.
			const WdgIf_InterfaceFunctionsType *WdgIfStateCombinerFunctions Pointer to the functions of the watchdog device driver connected to the master.
			WdgIf_StateCombinerSharedMemory *WdgIfStateCombinerSData Pointer to the shared memory.
			const

			WdgIf_StateCombinerSlaveTriggerPatternType Type **WdgIfStateCombinerSlaveTriggerPattern Pointer to an array of data for State Combiner configuration. One element for each slave.

Table 5-2 State Combiner Type Definitions

5.3 Services provided by WdgIf

5.3.1 WdgIf_SetMode

Prototype	
Std_ReturnType **WdgIf_SetMode** (uint8 DeviceIndex, WdgIf_ModeType Mode)	
Parameter	
DeviceIndex	Identifies the watchdog instance
Mode	WDGIF_OFF_MODE: Watchdog disabled WDGIF_SLOW_MODE: Long timeout period (slow triggering) WDGIF_FAST_MODE: Short timeout period (fast triggering)
Return code	
Std_ReturnType	E_OK: API finished successfully E_NOT_OK: An error occurred during execution
Functional Description	
This function maps the SetMode service to the corresponding physical watchdog implementation according to the parameter DeviceIndex.	
Particularities and Limitations	
> Service ID: see table 'Service IDs' > This function is synchronous. > This function is non-reentrant.	
Expected Caller Context	
> This service is expected to be called in task context.	

Table 5-3 WdgIf_SetMode

5.3.2 WdgIf_SetTriggerCondition

Prototype	
Std_ReturnType **WdgIf_SetTriggerCondition** (uint8 DeviceIndex, uint16 Timeout)	
Parameter	
DeviceIndex	Identifies the watchdog instance
Timeout	Timeout value in milliseconds for setting the trigger
Return code	
Std_ReturnType	E_OK: API finished successfully E_NOT_OK: An error occurred during execution
Functional Description	
This function maps the SetTriggerCondition service to the corresponding physical watchdog according to the parameter DeviceIndex.	

Particularities and Limitations
> Service ID: see table 'Service IDs' > This function is synchronous. > This function is non-reentrant.
Expected Caller Context
> This service is expected to be called in task context.

Table 5-4 WdgIf_SetTriggerCondition

5.3.3 WdgIf_SetTriggerWindow

Prototype		
Std_ReturnType **WdgIf_SetTriggerWindow** (uint8 DeviceIndex, uint16 WindowStart, uint16 Timeout)		
Parameter		
DeviceIndex	Identifies the watchdog instance	
WindowStart	Minimum time until next watchdog service is allowed in milliseconds	
Timeout	Timeout value in milliseconds for setting the trigger	
Return code		
Std_ReturnType	E_OK:	API finished successfully
	E_NOT_OK:	An error occurred during execution
Functional Description		
This function maps the SetTriggerWindow service to the corresponding physical watchdog according to the parameter DeviceIndex.		
Particularities and Limitations		
> Service ID: see table 'Service IDs' > This function is synchronous. > This function is non-reentrant.		
Expected Caller Context		
> This service is expected to be called in task context.		

Table 5-5 WdgIf_SetTriggerWindow

5.3.4 WdgIf_GetVersionInfo

Prototype	
void **WdgIf_GetVersionInfo** (Std_VersionInfoType* VersionInfoPtr)	
Parameter	
VersionInfoPtr	Pointer to where to store the version information of this module

Return code	
--	--
Functional Description	
WdgIf_GetVersionInfo returns the version information of this module.	
Particularities and Limitations	
> Service ID: see table 'Service IDs' > This function is synchronous. > This function is non-reentrant. > This function is only available if preprocessor switch WDGIF_VERSION_INFO_API set to STD_ON.	
Expected Caller Context	
> This service is expected to be called in task context.	

Table 5-6 WdgIf_GetVersionInfo

5.4 Services used by WdgIf

In Table 5-7 services provided by other components, which are used by the WdgIf are listed. For details about prototype and functionality refer to the documentation of the providing component.

The external functions must not degrade the quality level of the WdgIf. Hence, the possibility to use wrapper functions is provided so that either:

> the wrapper function calls the external function (e.g. context switch), or

> the wrapper function provides an alternative implementation of the external function.

Note
In case of using wrapper functions, these must be implemented according to the required quality level of the system (e.g. ASIL D).

All wrapper functions have the prefix "Appl_".

Component	Function	Description
OS	GetSpinlock() / ReleaseSpinlock()	If the State Combiner functionality is used (preprocessor option WDGIF_USE_STATECOMBINER is STD_ON) and if the preprocessor option WDGIF_STATECOMBINER_USE_OS_SPIN_LOCK is STD_ON, these OS functions are used in order to synchronize the State Combiner instances running on different processor cores. The declaration is included with Os.h. Note: If these functions do not meet the target quality level of the system, then the wrapper functions Appl_GetSpinlock() and

		`Appl_ReleaseSpinlock()` must be used. Note: These functions use the spinlock ID configured with the configuration parameter `WdgIfStateCombinerSpinlockID`. This spinlock must be initialized before the WdgIf is invoked for the first time (i.e. the overlying WdgM main function is invoked for the first time after system start-up).
`Appl_Spinlock`	`Appl_GetSpinlock()` / `Appl_ReleaseSpinlock()`	If the State Combiner functionality is used (preprocessor option `WDGIF_USE_STATECOMBINER` is `STD_ON`) and if the preprocessor option `WDGIF_STATECOMBINER_USE_OS_SPIN_LOCK` is `STD_OFF`, these user defined functions are used in order to synchronize the State Combiner instances running on different processor cores. The expected declarations are included with `Appl_Spinlock.h: Std_ReturnType Appl_GetSpinlock (uint32 ID); Std_ReturnType Appl_ReleaseSpinlock (uint32 int ID);` Note: These functions use the spinlock ID configured with configuration parameter `WdgIfStateCombinerSpinlockID`. This spinlock must be initialized before the WdgIf is invoked for the first time (i.e. the overlying WdgM main function is invoked for the first time after system start-up).

Table 5-7 Services used by the WdgIf

6 Configuration

This section describes the configuration of the WdgIf. Only link time configuration is used for the WdgIf.

6.1 Configuration Variants

The WdgIf supports the configuration variants

> `VARIANT-PRE-COMPILE`

The configuration classes of the WdgIf parameters depend on the supported configuration variants. For their definitions please see the WdgIf_bswmd.arxml file.

The WdgIf can be configured using the following tool:

> DaVinci Configurator 5 (AUTOSAR 4 packages only). Parameters are explained within the tool.

The outputs of the configuration and generation process are the configuration source files.

6.2 Integration with MICROSAR / fully AUTOSAR compliant Wdg drivers

In order to integrate the WdgIf with a MICROSAR / fully AUTOSAR-compliant watchdog driver the `WdgIfDevice` must be configured as following:

Driver-API as specified by AUTOSAR:

> Only the AUTOSAR parameter `WdgIfDriverRef` has to be configured by referencing either a watchdog driver directly (`WdgGeneral`) or a `WdgIfStateCombinerMaster` or `WdgIfStateCombinerSlave`. All other parameters (`WdgIfDeviceIncludeFile`, `WdgIfDeviceSetMode` and `WdgIfDeviceSetTriggerCondition`) will be updated automatically. If a `WdgIfStateCombinerSlave` is referenced the parameters mentioned above does not have to be configured.

Driver-API as not specified by AUTOSAR:

> The parameters (`WdgIfDeviceIncludeFile`, `WdgIfDeviceSetMode` and `WdgIfDeviceSetTriggerCondition`) must be configured to satisfy the driver's need.

Note
If the WdgM is the caller of the WdgIf (i.e. function `WdgIf_SetTriggerWindow()` is used to service the watchdog device), the parameter `WindowStart (WdgMTriggerWindowStart)` has no effect, because it cannot be passed to an AUTOSAR-compliant driver. It is then good practice to set it to 0, because this would be the functional meaning of its absence.

6.3 Configuring the State Combiner

The State Combiner allows to configure the reference cycle and the expected counter increments interval for each slave.

- Allows the user to determine and configure the values for reference cycle and number of expected triggers per trigger. Can be used to optimize reaction time.
- Does not allow changing the master or slave period unless the ratio between them stays the same.
- The State Combiner checks whether the number of slave triggers corresponds to the configuration – the system integrator must make sure that the configured values are correct!

6.3.1 Configuration for Synchronous Mode

In order to configure the State Combiner for synchronous mode following parameters must be configured in the ECU description:

- Set `WdgIfUseStateCombiner` to `true` (enable State Combiner).
- Set `WdgIfStateCombinerReferenceCycle` to the expected number of slave triggers.
- Set `WdgIfStateCombinerSlaveIncrementsMin` to the constant number of expected slave triggers.
- Set `WdgIfStateCombinerSlaveIncrementsMax` to the constant number of expected slave triggers as well.

> **Note**
> The last three parameters are set for each slave.

> **Note**
> `WdgIfStateCombinerSlaveIncrementsMin` and `WdgIfStateCombinerSlaveIncrementsMax` must have the same value for synchronous mode!

Example scenario 1: Assume that the necessary conditions for synchronous mode apply, the master period is 20ms and the slave period is 20ms. The following configuration is recommended for the State Combiner:

- `WdgIfUseStateCombiner = true`
- `WdgIfStateCombinerReferenceCycle = 1`
- `WdgIfStateCombinerSlaveIncrementsMin = 1`
- `WdgIfStateCombinerSlaveIncrementsMax = 1`

Example scenario 2: Assume that the necessary conditions for synchronous mode apply, the master period is 20ms and the slave period is 40ms. The following configuration is recommended for the State Combiner:

> `WdgIfUseStateCombiner = true`
> `WdgIfStateCombinerReferenceCycle = 2`
> `WdgIfStateCombinerSlaveIncrementsMin = 1`
> `WdgIfStateCombinerSlaveIncrementsMax = 1`

Example scenario 3: Assume that the necessary conditions for synchronous mode apply, the master period is 30ms and the slave period is 10ms. The following configuration is recommended for the State Combiner:

> `WdgIfUseStateCombiner = true`
> `WdgIfStateCombinerReferenceCycle = 1`
> `WdgIfStateCombinerSlaveIncrementsMin = 3`
> `WdgIfStateCombinerSlaveIncrementsMax = 3`

6.3.2 Configuration for Asynchronous Mode

If the State Combiner is configured for asynchronous mode, then the reference cycle and the maximum and the minimum numbers of expected slave triggers are entered as part of the configuration. Following needs to be configured:

> `WdgIfUseStateCombiner` to `true` (enable State Combiner).
> `WdgIfStateCombinerReferenceCycle` to the required value.
> `WdgIfStateCombinerSlaveIncrementsMin` to the required value.
> `WdgIfStateCombinerSlaveIncrementsMax` to the required value.

Note
The last three parameters have to be set for each slave.

Example scenario:

Assume that the necessary conditions for asynchronous mode apply, the master period is 20ms and the slave period is 20ms. Jitter for both master and slave is maximum 2ms. Following configuration is optimal for the State Combiner:

> `WdgIfUseStateCombiner = true`
> `WdgIfStateCombinerReferenceCycle = 2`
> `WdgIfStateCombinerSlaveIncrementsMin = 1`
> `WdgIfStateCombinerSlaveIncrementsMax = 3`

7 Glossary and Abbreviations

7.1 Glossary

Term	Description
<infix>	A placeholder with this name is interpreted as follows: > In case of AUTOSAR 4 compatible environment the <infix> placeholder consists of the vendor ID and device name string of the configured Watchdog driver. > In case of AUTOSAR 3 compatible environment the <infix> placeholder consists of the device name string of the configured Watchdog driver.
Check interval	The duration between two consecutive points in time when the master checks a slave trigger counter. It is the reference cycle multiplied by the master invocation period.
Master	State Combiner instance which is configured to work in master mode.
Slave	State Combiner instance which is configured to work in slave mode.
Master / Slave invocation	In the WdgM Stack, this is the point in time when the WdgM_MainFunction of the overlying WdgM is invoked – this main function eventually calls the master / slave.
Reference cycle	The number of master periods between two consecutive checks of the slave by the master. One means that the master checks a slave each time the master is invoked; two means that the master checks a slave every second time the master is invoked, etc. Note that for each slave the reference cycle can be different. See also check interval.
Slave trigger errors	They are: > slave invocation omissions, > slave invocation drifting, > too frequent slave invocations and > unscheduled triggers.
Trigger counter	A variable in shared memory for each slave which starts from 0 and is being incremented by its slave each time the slave is invoked with a trigger request.
Number of slave triggers	The number of trigger requests of a slave during a given period of time.

Table 7-1 Glossary

7.2 Abbreviations

Abbreviation	Description
API	Application Programming Interface
AUTOSAR	AUTOSAR (AUTomotive Open System ARchitecture) is a worldwide development partnership of car manufacturers, suppliers and other companies from the electronics, semiconductor and software industry.
DEM	Diagnostic Event Manager
DET	Development Error Tracer
ECU	Electronic Control Unit
MCU	Microcontroller Unit
WCD	Worst Case Delay
Wdg	Watchdog Driver
WdgIf	Watchdog Interface
WdgM	Watchdog Manager

Table 7-2 Abbreviations

8 Contact

Visit our website for more information on

> News
> Products
> Demo software
> Support
> Training data
> Addresses

www.vector.com